LiU-ITN-TEK-A-13/030--SE

Automatiserat system för att skapa unika vinjetter

Åsa Kjäll

2013-06-13

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

LiU-ITN-TEK-A-13/030--SE

Automatiserat system för att skapa unika vinjetter

Examensarbete utfört i Medieteknik
vid Tekniska högskolan vid
Linköpings universitet

Åsa Kjäll

Handledare Stefan Gustavson
Examinator Matt Cooper

Norrköping 2013-06-13

Automated system to create unique vignettes

Åsa Kjäll

June 14, 2013

Abstract

This document describes the work for a master's thesis in Media Technology and Engineering at Linköping University. The work consisted of building a system for defining and creating vignettes for horse racing shows. The vignettes are built up by a montage of video clips which are changed with the seasons, time of day and betting product. The system randomizes the video files and the audio files depending on the season, time of day and current betting product when creating the vignettes. Each vignette will therefore be unique. The work also consisted of implementing the vignette graphics in the software *Ventuz*.

Contents

Chapter 1

Introduction

1.1 Definition of terms

In this thesis the following list of terms will be used.

Vignette Television show intro.

Vignette system The implemented system for vignette creation.

User Person who is using the system, either to create a new vignette or to create a new vignette definition.

Vignette definition Definition of vignette type.

Leg Some betting products, like *V75*, consists of a number of legs. Each day several races are held, and the races belonging to a particular betting product are called legs.

1.2 Background

In Sweden trotting is a popular sport with races 364 days a year. Each day several races run simultaneously on different tracks, and each year about 10.000 races are held. ATG's underlying company, Kanal 75, covers all of these races and makes productions for TV4 and ATG's own channel, ATG 24. These productions, like most others, begin with a vignette. Since Kanal 75 is covering all of these races, a lot of vignettes need to be made. At the moment these vignettes are made in *After Effects*, which takes time because every time a change to the appearance of the vignette needs to be made, a new vignette needs to be rendered. For the graphics which is used together with the vignettes, software called *Ventuz* is used. *Ventuz* is a real-time rendering 3D-graphics software. The purpose of this thesis is to move the creation of vignettes from *After Effects* to *Ventuz*, saving time in rendering. The system will not only be used to make vignettes, but also to make video

montage. The system therefore needs to be general to enable easy and fast changes.

1.3 System requirements

This thesis work will be about making an automated system for creating unique vignettes, at the latest one hour before the broadcast, and in a maximum of 15 minutes. The total time to make a vignette should be about 80% less than with present-day methods. The main focus will be on making a robust system that will n̈everf̈ail. Since this is for live television the system always has to work. Also the system needs to be general so making changes to the vignette is an easy task. The performance of the system is also important, again since the system is meant for live television the graphics should never freeze.

1.4 System Limitation

To keep within the time frame, and focus on the important aspects of the thesis work, some limitations were set on the vignette system.

Not all the internal filters from *Avisynth*, see section 2.3 are implemented, only the most relevant to this thesis work, and not all of the implemented filters are available in the Graphical user interface, see section 2.6. This thesis work is focused on building the system, and not making it completely ready for production. Kanal 75's art director and the people working with graphics development have been trained to use the system, but not the people who control the graphics at live broadcasting.

1.5 Betting products at ATG

Listed below are the ATG's betting products. Each betting product belongs to a betting color. The color of the current betting form is listed in the description.

V75 The most popular betting form. Bet on horses in seven legs. Five, six and seven correct winners give payouts. Blue color.

V86 Bet on horses in eight legs. Six, seven and eight correct winners give payouts. Purple color.

V65 Bet on horses in six legs. Five and six correct winners give payouts. Red color.

V64 Bet on horses in six legs. Four, five and six winners give payouts. Orange color

V5 Bet on horses in five legs. Only five correct winners gives a payout. Turquoise color.

V4 Bet on horses in four legs. Only four correct winners gives a payout. Green color.

Daily Double and Lunch double Bet on horses in two designated legs. Lunch double is available at lunch and daily double is available with V75, V86 and V65. Turquoise color.

Trifecta Bet on horses in one leg. Find the horses that finish first, second and third. Turquoise color.

Quinella Bet on horses in one leg. Find the horses that finish first and second. Turquoise color.

Exacta Bet on horses in one canter race. Find the horses that finish first and second. Turquoise color.

Win and show In win, bet on horses in one leg. Find the horse that finishes first. In show, bet on horses in one leg. Find the horses that finish first, second or third. Turquoise color.

At [1] more information about the betting forms is available.

1.6 Current vignette types

The current vignette types which Kanal 75 uses for their productions are as follows.

Vignette - Television show intro. Contains five video segments where the four first have a duration of 60 frames, or 2.4 seconds[1]. The last segment can be of any length, but needs to be longer than 250 frames to get past the minimum vignette length. When the video clips are put together a dissolve with a duration of 12 frames is used.

To commercial bumper - A shorter type of vignette, called a bumper. This type of bumper is used when the show is going to commercial. It consists of two video segments, one with a duration of 60 frames and one with a duration of at least 250 frames. When the video clips are put together a dissolve with duration of 12 frames is used.

From commercial bumper - Similar to the previous bumper, but with different audio. When the video clips are put together a dissolve with a duration of 12 frames is used.

[1] 60 frames divided by 25 frames per second

Short bumper - A shorter type of bumper which contains only one video segment with a duration of at least 250 frames.

General bumper - Same as the commercial bumpers, but with different audio. When the video clips are put together a dissolve with a duration of 12 frames is used.

Leg bumper - Each leg has its own bumper. V75 consists of seven legs and therefore has seven leg bumpers.

1.7 Software introduction

This thesis work has mainly been developed in the software system, *Ventuz*. Ventuz is developed in Germany and was established in 2004. *Ventuz* is a real-time 3D-graphics creator. It is used for presentations, interactive environments and TV broadcasting. At Kanal 75 it is used mainly for TV broadcasting. It is a node-based drag and drop programming environment. Simple geometry, such as planes, cubes and spheres, is implemented in *Ventuz*. For more advanced geometry a mesh loader node is available. Animation can be imported together with the imported geometry. Non-linear animations can be made inside *Ventuz*. The animations can be state-based and event driven. *Ventuz* can be connected to databases, *XML* files, text files and *Excel* files for data fetching. Simple expression nodes, math effect nodes and other logic can be used to build up the graphics environment. If these nodes do not provide the desired functionality, *C#* and *VB.NET* nodes are available for more advanced functions. *HLSL* nodes provide the ability to do shader programming. *Ventuz* has a frame rate of 25 frames per second. Figure 1.1 shows an image of the *Ventuz* environment. In the top left window the graphics hierarchy is built. The nodes added to this window are the ones that will be rendered and shown as output. The graphics objects are rendered from top to bottom. In the top center window, called the content window, the appearance and functionality of the graphics is set up. The *C#* and *VB.NET* nodes, math effect nodes, animation nodes and other logic nodes are dragged and dropped in this window. On the right side of the content window is the property window. When selecting a node, its properties are shown in this window. The *C#* node is selected in figure 1.1 and in the property window the script's input and output parameters are shown. In the bottom left window the toolbox is found. The toolbox contains all the available nodes, grouped into categories. This window also contains the animation timeline. In the bottom right window, called the render window, the output from *Ventuz* is shown. In this window the changes made in the scene are shown in real time.

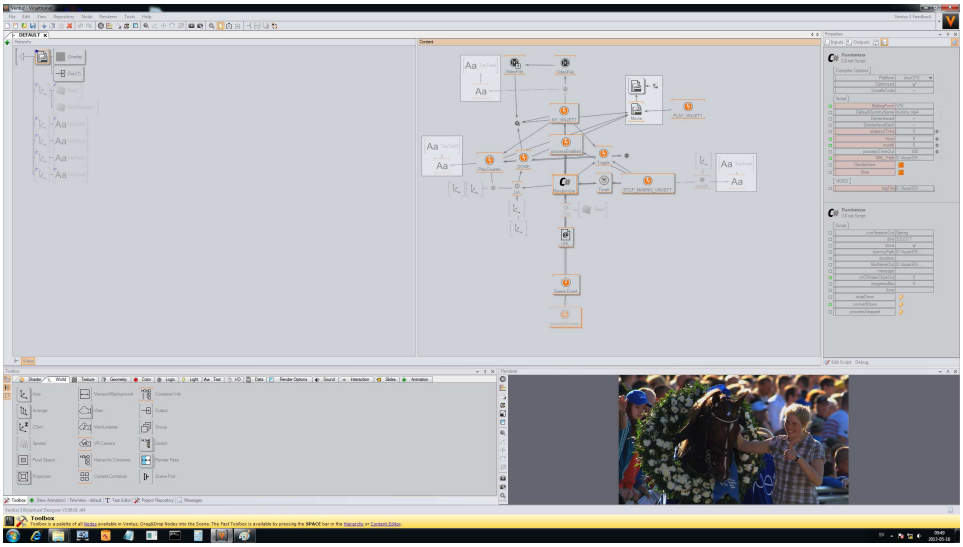

Figure 1.1: The *Ventuz* environment.

1.8 Thesis outline

Chapter 2: The vignette system This chapter will explain the process of developing the vignette system.

Chapter 3: The vignette graphics The vignettes' graphics will be explained in this chapter. How the graphics were implemented and what function they provide.

Chapter 4: Conclusions and future work The final chapter of this thesis will discuss the conclusions made throughout the development process. Thoughts about future work to the vignette system will also be discussed.

Chapter 2

The vignette system

2.1 Ventuz and Codec

One of the most important parts of this thesis work is to get as good performance as possible. At Kanal 75 they have tried to build a similar system before, but the performance was not good enough. Both the video and the audio suffered from latency issues from time to time. In the *Ventuz* user manual they describe that movie playback is a possible performance issue. Most videos are encoded to a compressed format, which take up less space than the uncompressed video. To play a compressed video it needs to be decoded first into its original format. This could be an issue, because decoding can sometimes be CPU intensive. Decoding is in *Ventuz* the first step on the CPU when a video is to be played. *Ventuz* has two nodes for movie playback, the *Movie clip* node and the *Advanced movie clip* node. The *Movie clip* node uses *DirectShow* to decode the video files and the *Advanced movie clip* node uses the *FFMpeg* project [9] decoders. The latter decoders have better performance and are less CPU intensive, than the *DirectShow* decoders. The *Advanced movie clip* node was chosen to be tried first because it is said to give better decoding performance. The second step on the CPU in *Ventuz* is that the decoded data needs to be copied to a *DirectDraw* texture in 3D space. This step is what makes *Ventuz* have possible performance issues with movie playback compared with other video players. Other video players copy the decoded data to 2D overlay planes. There are different types of encoders and decoders. One way to get better performance is to find a codec, (enCOder/DECoder), which is less CPU intensive to decode. The first part of the thesis work is to try out different codecs. When a codec is chosen, a suitable container for that codec needs to be chosen as well.

The important aspects of choosing a video codec are the quality, bitrate, file size and compatibility with *Ventuz*. Finding a suitable workflow for collecting and replacing video clips for the vignette system is also an important aspect. There cannot be too many steps from finding the video

clips and having them in a *Ventuz*-friendly codec. The main goal is not to get the highest quality of the video, but the highest visual quality. It is to get the quality that is perceived as high quality, but at lower bit-rate and video file size. The resolution of the video files needs to be 1920x1080[1], which means the codec has to have support for 1920x1080 HD resolution.

A few limitations were set on which video codecs to compare, in order not to spend too much time on finding a suitable codec. The chosen codecs were *H.264*, *Xvid*, *DivX* and *DNxHD*. The first three were chosen because they are widely used at Kanal 75 already, and they are mentioned in the *Ventuz* user manual as working codecs. They are also mentioned on the *Ventuz* forum as fairly good working codecs. The last one, *DNxHD*, is an *Avid* [14] codec. The *Avid* system is used by Kanal 75 for video editing. All these codecs have support for 1920x1080 HD resolution and they are lossy compression formats, which means that some quality is lost in the encoding/decoding process. The benefit of a lossy compression format is that the compressed file takes up less space, than a lossless[2] compression format.

2.1.1 DivX

The *DivX* codec is developed by *DivX Inc.* It is a popular video codec, but it is a commercial product, and therefore it is not entirely free to use. *DivX* supports multi-thread encoding and decoding, which can be used on multiple core, multiple processor, or hyperthreaded systems to achieve better performance.

2.1.2 Xvid

Xvid is an open source video codec which is similar to *DivX*. It is also supports multi-threading. The codec has four main profiles; mobile, portable, home and highdef. The mobile profile is developed for devices with small displays, such as mobile phones. The portable profile supports video up to VGA resolution. PAL[3] resolutions are supported in the home profile, which is developed for home video devices. Finally the highdef profile is used for HD resolutions.

2.1.3 H.264

H.264 is a video codec standard which was published in 2003. It was developed with the aim to achieve better performance than the previous standards. The standard has more efficient compression than its predecessors

[1] 1920x1080 is HD resolution, which is required for HD broadcasting
[2] Lossless means that no quality lost during the encoding/decoding process
[3] European TV standard.

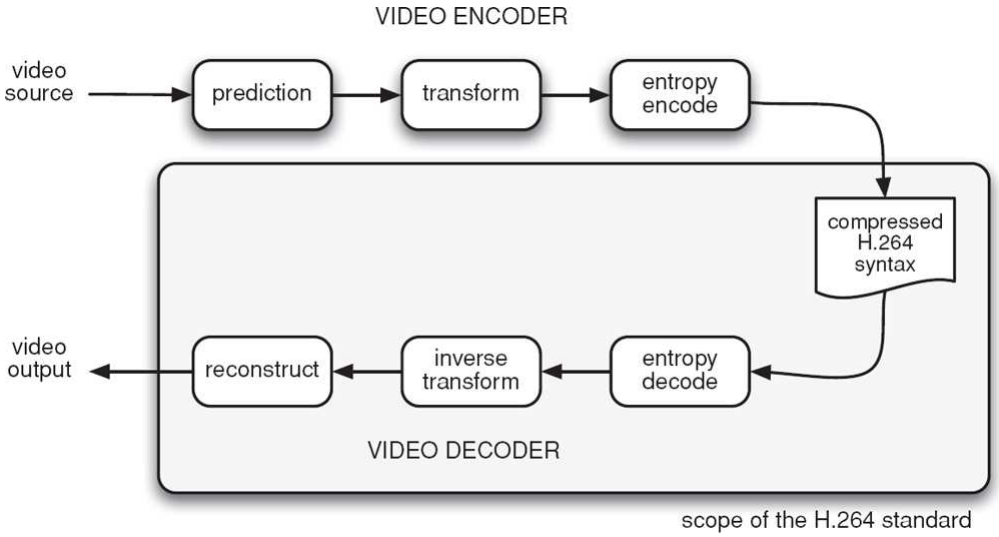

Figure 2.1: *H.264* Encoding/Decoding process, from [13]

and therefore video compressed with this standard will take up less space than with the earlier standards. Figure 2.1 shows the *H.264* encoding and decoding process. To learn more about the *H.264* standard [13] is a good source of information.

2.1.4 DNxHD

DNxHD is developed by *Avid Technology Inc.* The codec is part of the *VC-3* standard, developed by *SMPTE*, Society of Moving Picture and Television Engineers [14]. *VC-3* is a standard for compression formats. Even though *DNxHD* is a commercial product it is now available free of charge. *DNxHD* has five profiles, or families as Avid calls them. All of them support HD resolution, the differences between them are mainly which color space they are using.

2.1.5 Codec evaluation

H.264, *Xvid* and *DivX* are part of the *MPEG-4* standard. *MPEG-4* is a standard for compression formats. The standard is developed by *MPEG* (Moving Picture Experts Group)[15]. *Xvid* and *DivX* both belong to *MPEG-4* part 2, Visual Advanced Simple Profile, ASP, while *H.264* belongs to part 10. The *MPEG-4* part 10, Advanced Video Coding is a standard for more efficient codecs. Between *Xvid* and *DivX*, *Xvid* would be chosen for this thesis. They are both part of the same standard, which means they have more similarities than differences. The great advantage of *Xvid* is that it is open source. According to [11] neither *Xvid* nor *DivX* could match *H.264*

in bit-rate. *H.264* was especially efficient in video with complex content. [11] compares *H.264* with *Xvid* and *DivX* by encoding six different video clips with each of these codecs[4] and using two methods to compare the results. The two methods used are PSNR-Y, peak signal-to-noise ratio [21], and JND, just noticeable difference [20]. Using the PSNR-Y method, *H.264* was able to achieve up to 50% bit rate savings in comparison to *DivX*. *H.264* was chosen over the other two, partly because of the results of [11]. Another, but not as important, reason why *DivX* did not get chosen was because it was difficult to find information about the codec. Even on *DivX's* website it was difficult to find relevant information about the codec.

DNxHD was not chosen because it is an *Avid* codec. When *H.264* provides good quality with good performance there is no need to choose a less widely used codec.

2.1.6 Containers

Only a few container formats was looked into. Since *H.264* was chosen as a codec it was natural to choose a container that is also part of the *MPEG-4* standard. Since the *QuickTime* container, *.mov* is only available in 32-bit it was excluded from the container choices. Also on the *Ventuz* forum users have claimed that *QuickTime* does not work smoothly for *Ventuz*. The container *MP4* is part of the *MPEG-4* part 14 standard. It was chosen because it is a *MPEG-4* standard, and because it was recommended on the *Ventuz* forums.

2.1.7 Summary

Although these choices were made for this thesis work, the system was to be built in a manner that would make it easy to choose differently. These codecs and container choices are mainly made for this thesis. The world of multimedia is always changing, and new standards are available at regular intervals. It would be unwise to build the system without the ability to make different choices later.

2.2 First approach

The vignettes are, as mentioned before, built up from several video clips put together into a single video clip with audio added to it. The audio could be built up from shorter audio parts into one longer audio file. In *Ventuz* the *Audio clip* nodes have a 'completed' event property which is triggered when the audio clip has finished playing. A test was made where one *Audio clip* node's 'completed' event was connected to another *Audio clip* node's 'play'

[4] And a few others, but they are irrelevant to this thesis.

event. This solution gave pauses between the audio clips which were not acceptable. The *Key frame animation* node was tested next. Two *Audio clip* nodes were used and the second *Audio clip* node's 'play' event was triggered on the same frame as the first *Audio clip* was completed, see figure 2.2.

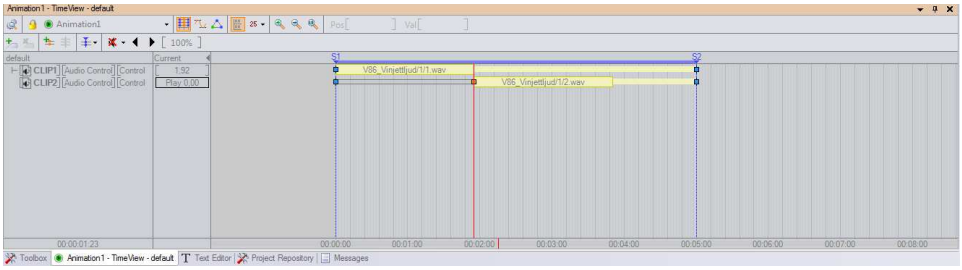

Figure 2.2: The audio clips in the animation timeline.

Glitches could be heard in the output audio file, and when the same audio clips were put together using other software the output file did not contain the same number of frames as in *Ventuz*. This solution also gave results which were not acceptable. When putting the video clips together in *Ventuz*, two methods were tested. The *Movie clip* node has a built-in cue system which cues the files in a specified folder. Early tests showed that the cue system did not output seamless exchanges of the video clips. Also, using the built-in cue system made it impossible to have a dissolve between the clips. The other method was to use the *Key frame animation* node. The *Movie clip* node's 'play' control was therefore connected to the *Key frame animation* node, see figure 2.3.

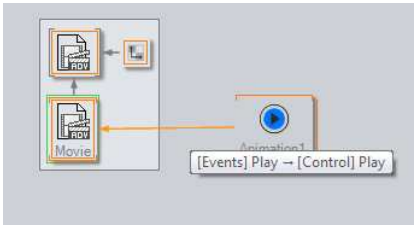

Figure 2.3: The connection between the video clip node and the *Key frame animation* node.

Using the *Key frame animation* a dissolve between the video clips could be implemented. This method worked quite nicely but, since the *Key frame animation* node did not work for the audio, it was decided to take the whole process of putting video clips and audio clips together and do it outside of *Ventuz*. Since *Ventuz* is focused on as a real time, 3D graphics creation

software, taking the process of editing video and audio outside *Ventuz* allows *Ventuz* to use its power and performance on the graphics of the vignette instead.

2.3 Avisynth

Since it was decided to take the process of editing video and audio outside *Ventuz*, a video and audio editing software needed to be found. Preferably free, open source software. After a tip from a colleague, the software *Avisynth* was looked into. *Avisynth* is free, open source software for video and audio editing. It works as a frameserver, which means it provides other applications with video and audio, but without the need for temporary files. The video application receiving the video from the frameserver sees the incoming video as a small uncompressed file, while the video really could be a highly compressed video. *Avisynth* uses a script-based system for video and audio editing. It comes with a large number of filters and functions, and also the ability to build your own functions. The filters and functions are used to manipulate the audio and video files. It is easy to add video files together into a longer video file and add audio files together into a longer audio file. A simple filter is used to add audio to a video file. *Avisynth* is a powerful tool for non-linear editing and it is easy to learn. An example script can be found in A.1. The script is saved as a *.avs* file and can be run in for example *Windows Media Player*.

A few tests were made to see if the software would be suitable for this thesis. Since the script based system provides a general solution, and both video and audio are put together seamlessly, without any latency issues, this was a perfect choice for video and audio editing. *Avisynth* provides many filters and functions, and the possibility to build your own functions. In *Ventuz* there are no video editing filters, which makes *Avisynth* a more general solution. For this project a few of the *Avisynth* filters and functions were implemented in the system, and the architecture of the system was designed to make it easy to add more filters or functions if needed (see section 2.5.1).

2.3.1 Filters and functions

The internal filters of *Avisynth* are categorized into the following groups: *Media file filters*, *Color conversion and adjustment filters*, *Overlay and mask filters*, *Geometric deformation filters*, *Pixel restoration filters*, *Timeline editing filters*, *Interlace filters*, *Audio processing filters*, *Conditional and other Meta filters*, and finally *Debug filters*. The filters described in the following sections are those which are implemented in the vignette system. The description tells how the filters are implemented, and may differ from how the

filters are described on *Avisynths's* webpage. To better suit the vignette system the filters may be scaled down, that is they may have fewer parameters than described on *Avisynths's* webpage.

Media file filters

The *Media file filters* are filters used to open files for processing. The ones implemented in the vignette system for opening video are: *AviSource*, *DirectShowSource* and an external filter called *QTInput*. *AviSource* opens *avi* files, *QTInput* opens *QuickTime* files and *DirectShowSource* is used to open the remaining file formats. *AviSynth* can also open an image as input and 'play' it for a specified number of frames. The implemented filter for opening images is *ImageSource*. For opening audio files the filter *DirectShowSource* was also used. The final filter implemented from this group is named *Import* and it allows for other *Avisynth* scripts to be imported into the current script.

Color conversion and adjustment filters

The *Color conversion and adjustment filters* are filters applied to input files to change their color space and/or adjust the colors. Only one filter is implemented from this category, the *ConvertTo-filter*. The color formats which can be converted to are; RGB, RGB24, RGB32, YUY2, Y8, YV411, YV12, YV16 and YV24 [5][4].

colorformats = ...	Planar/Interleaved	Chroma resolution
RGB	Interleaved	full chroma - 4:4:4
RGB24	Interleaved	full chroma - 4:4:4
RGB32	Interleaved	full chroma - 4:4:4
YUY2	Interleaved	chroma shared between 2 pixels - 4:2:2
Y8	planar/interleaved	no chroma - 4:0:0
YV411	planar	chroma shared between 4 pixels - 4:1:1
YV12	planar	chroma shared between 2x2 pixels - 4:2:0
YV16	planar	chroma shared between 2 pixels - 4:2:2
YV24	planar	full chroma - 4:4:4

Table 2.1: Information about the color spaces in *Avisynth*, from [4]

[5]For information on color spaces, see chapter 2.4 in [13]

Overlay and mask filters

The *Overlay and mask filters* are filters used to layer video clips or images with other video clips or images. The layering could be made with or without an alpha-mask. The vignette system has one mask filter and one overlay filter implemented. The *Mask filter* works as masks do in *Photoshop*. A white pixel in the mask clip or image corresponds to a fully opaque pixel, while a black pixel corresponds to a fully transparent pixel. The *Overlay filter* takes two files and overlays the first file with the second one. The *Overlay filter* has five optional parameters. The first parameters, which are specified together, are x and y. X and y specifies where the overlay clip is being placed on the background clip, that is to say x and y defines the offset of the overlay image on the original image. The third optional parameter is adding a mask filter, which will be applied to the overlay file. The fourth option is an opacity value, which will specify how transparent the overlay file will be. The fifth and final option is which mode will be used to overlay the files, and the options are: Blend, Add, Subtract, Multiply, Chroma, Luma, Lighten, Darken, SoftLight, HardLight, Difference and Exclusion. With the *Overlay filter* the two clips do not need to be of the same size, or the same color space. With the *Mask filter* however the files need to be converted to, if they are not already in, color space RGB32.

Mode	Description
Blend	This is the default mode. When opacity is 1.0 and there is no mask the overlay image will be copied on top of the original. Ordinary transparent blending is used otherwise.
Add	This will add the overlay video to the base video, making the video brighter. To make this as comparable to RGB, overbright luma areas are influencing chroma and making them more white.
Subtract	The opposite of Add. This will make the areas darker.
Multiply	This will also darken the image, but it works different than subtract.
Chroma	This will only overlay the color information of the overlay clip on to the base image.
Luma	This will only overlay the luminosity information of the overlay clip on to the base image.
Lighten	This will copy the light infomation from the overlay clip to the base clip, only if the overlay is lighter than the base image.
Darken	This will copy the light infomation from the overlay clip to the base clip, only if the overlay is darker than the base image.
SoftLight	This will lighten or darken the base clip, based on the light level of the overlay clip. If the overlay is darker than luma = 128, the base image will be darker. If the overlay is lighter than luma = 128, the base image will be lighter. This is useful for adding shadows to an image. Painting with pure black or white produces a distinctly darker or lighter area but does not result in pure black or white.
HardLight	This will lighten or darken the base clip, based on the light level of the overlay clip. If the overlay is darker than luma = 128, the base image will be darker. If the overlay is lighter than luma = 128, the base image will be lighter. This is useful for adding shadows to an image. Painting with pure black or white results in pure black or white.
Differece	This will display the difference between the clip and the overlay. Note that like Subtract a difference of zero is displayed as grey, but with luma=128 instead of 126. If you want the pure difference, use mode="Subtract" or add ColorYUV(off_y=-128)
Exclusion	This will invert the image based on the luminosity of the overlay image. Blending with white inverts the base color values; blending with black produces no change.

Table 2.2: Information about the different modes in the overlay filter, from [8]

Geometric deformation filters

The filters that belong to this category are used to make changes to the video's or image's geometric properties. Five filters were implemented in the vignette system. The *Crop filter* is one of them, and it is used to crop pixels from the top, bottom and the left and right side of the video or image. A similar filter is the *ReduceBy2 filter*. It can either, reduce the horizontal or the vertical size by half, or it can reduce both the horizontal and the vertical size by half at once. The third filter in this category is the *Flip filter*. It either flips the video up-side-down or flips the video from left to right. There is also the *Rotate filter*, which rotates the video or image 90 degrees clockwise or counter-clockwise. The last of the implemented filters from this group is the *Resize filter*. There is a wide selection of resizing algorithms; BilinearResize, BicubicResize, BlackmanResize, GaussResize, LanczosResize, Lanczos4Resize, PointResize, SincResize, Spline16Resize, Spline32Resize and Spline64Resize[6]. Beside the choice of algorithm the filter also takes the target height and target width as parameters.

Timeline editing filters

The filters in this category are, as the name implies, used to edit the files with respect to time. To this category belongs the most important filter for the vignette system, the *Dissolve filter*. The *Dissolve filter* makes two video files or images fade in and out of each other, given a number of frames for the dissolve to last. This category also contains the second most important filter, the *Trim filter*. This filter allows for video clips that are not of the exact length required to be combined into the vignette. Trim specifies on which frame to start playing the clip, and for how many frames it will play. As an option to the dissolve filter there are three more filters, *FadeIn*, *FadeOut* and *FadeInOut*. These filters linearly fade to/from black. For *FadeIn* it fades from black, for *FadeOut* it fades to black, and for *FadeInOut* it fades from black at the beginning and fades to black at the end. Just as with the dissolve filter the fade will last for a specified number of frames. The *Loop filter* will make a video play over and over for a specified number of times. To change the file's frame rate there is a frames-per-second, *FPS*, filter. There are three ways to change the frame rate of a file, *AssumeFPS*, *ChangeFPS* and *ConvertFPS*. *AssumeFPS* keeps the frame count when changing the frame rate, making the video play faster or slower. *ChangeFPS* removes or copies frames to change the frame rate. The last one, *ConvertFPS*, tries to avoid both of the previous filters methods, inserting/dropping frames and playing faster/slower. This filter uses two commonly used, by other FPS converters, methods for changing the frame rate. Unfortunately there is no

[6]To read about the different resizing algorithms, see [5]

information on which the methods are. *SelectEven* and *SelectOdd* are filters for making an output with only even or odd frames. *Reverse* is a filter for playing the files in reverse. It is not a commonly used filter.

Pixel restoration filters

These filters can be used to make the video or image sharper or more blurry. No filter from this group was implemented in the vignette system.

Interlace filters

The filters of this type are used to deal with video parity. Both interlaced and progressive [7] video can be processed in *Avisynth*. *Avisynth* always assumes that the input is frame-based, because the video files do not normally contain information about frames or fields. If it is known that an input file is field-based, the filter *AssumeFieldBased* could be used. The corresponding filter is *AssumeFrameBased*. These filters assume bottom field first, which means even-numbered fields. If this is not true, the *Compliment parity filter* can be used. The *Compliment parity filter* changes the bottom fields to top fields, or the other way around if the video file is top fields first. If the input is an interlaced file, sometimes it needs to be deinterlaced. Two deinterlacing filters are implemented. The first one is the *Bob* deinterlacer. The *Bob* algorithm takes the input clip and linearly interpolates between the fields [7]. The second is the *Weave* deinterlacer [6].

Audio processing filters

The *Audio processing filters* are applied to the audio files, and contain similar filters as for the video but in a much more limited form. One filter from this group is implemented, and it is not really an audio processing filter. It is called *AudioDub* and it adds the audio files to the video stream for the output.

Conditional and other meta filters

This family of filters contains *Conditional filters*, which at each frame evaluates some condition and outputs the corresponding option. The *Meta filters* are used together with other filters as an extra control. One filter from this family is implemented as part of an external function. The *Animation filter* is used in a function called *Zoom*. The *Zoom function* zooms a video or image during a specified number of frames. It enlarges the input file a specified number of times. The *Zoom function* can be found in A.2, together with a more detailed explanation.

[7] For information about interlaced and progressive video, see [13], at 2.3.3

Debug filters

These filters are mainly used for debugging, and are not implemented in the vignette system.

Summary

[3] contains a full list of the internal filters of *Avisynth*. There are also external filters, made by *Avisynth* users. These were not taken into consideration, with the exception of *QTInput*, because the scope of this thesis would then be too large. Instead focus was put on designing a structure that would ease the process of later adding filters and functions.

2.4 FFMPeg

Ventuz 64-bit is not able to read *.avs* files, and even if it did it would be a good idea to actually create a video file [8] for movie playback due to stability reasons. Also creating a file from the *Avisynth* script opens up the possibility to have the input video files in any codec and container format, since it is converted to an *H.264* codec in *MP4* container at this step. According to [19] *x264* is the best choice for encoding video to *H.264*. [19] has done thorough test on the different encoders and *x264* has won awards for being the best *H.264* encoder. This encoder was an obvious choice for encoding videos in this thesis work.

Ventuz uses *FFMPeg* to decode the video files for movie playback[9]. *FFMPeg* is a software system used to encode, decode, transcode, play etc. almost every multimedia format available. *FFMPeg* provides different tools for different purposes. For converting files in-between formats *FFMPeg* provides a command line tool, *ffmpeg*. Fortunately *ffmpeg* uses *x264* to encode video files to *H.264*, and also to decode *H.264*, which makes *FFMPeg* the obvious choice of encoding software [9]. The fact that *ffmpeg* can convert almost any format into *H.264* makes the choice even easier to make.

2.4.1 x264

x264 has two settings which are implemented in the vignette system. The first setting decides what quality the output video will be. This setting is called *CRF*, or constant rate factor. The values range from 0-51, where 51 is the worst quality and 0 is lossless. 18 correspond to visually lossless. With this method each frame will receive at least the bitrate required to achieve the desired quality.

[8] Since *Avisynth* only fakes an *AVI* file.

[9] If the *Advanced movie clip* node is used

Encodingspeed	time (s)	file size (MB)
ultrafast	17	1008
superfast	43	980
veryfast	43	982
faster	52	980
fast	80	980
medium	100	977
slow	178	930
slower	300	900
veryslow	420	863

Table 2.3: Encoding speed compared to file size.

The second setting is how fast the encoding will be. With faster encoding the compression rate will be worse, which will lead to a bigger output file. The guideline is to choose the slowest encoding you have time for. The encoding speed presets are: ultrafast, superfast, veryfast, faster, fast, medium, slow, slower, veryslow and placebo [10]. The *x264* encoding guide [10] says that placebo should be ignored since the encoding time is much higher than for veryslow, and will give pretty much the same result as veryslow. The typical *ffmpeg* command line for converting to *H.264* with *.mp4* container in this thesis would look like:

ffmpeg -i ”VideoFile.avs” -vcodec libx264 -preset veryfast -crf 18 -y ”VideoFile.mp4”

In table 2.3 a typical vignette with five segments were encoded using the different encoding speed presets. The *CRF* value is 18 in all cases.

2.5 System implementation

The vignettes are built up from a montage of video clips and audio clips. It can be any number of video clips and any number of audio clips. Making changes to the appearance of the vignettes is required to be an easy task. The idea is that the folders holding video clips and audio clips for the vignette are specified and the system orders the folders at random. From each folder video files are chosen at random and put together into the vignette. This will make every new vignette unique, since different clips are chosen at random every time the system is run. The vignettes change with the seasons to prevent having winter footage in the middle of the summer. They also change depending on the hour of the day. If it is daytime, day footage is used and if it is evening, evening footage is used. Each betting product has its own associated footage as well. A folder structure is defined

to get the correct footage of the season, the current betting product and the hour of the day. A general structure of the vignette elements needs to be designed and, every time the vignette appearance needs to be changed, a new definition is created which is based on the general structure. An important aspect of designing the structure is to ease the process of adding and removing elements.

2.5.1 System structure

Folder structure

The folder structure is hierarchical. At the top level are the category folders, which are the folders the user can choose from. For the vignettes there are at the moment five different folders at this level. The first contains close-up footage of horses. The second contains close-up non-horse footage. The third contains long shots of horses, and the fourth contains long shots of non-horse footage. The fifth and last contains extra-long footage, which can be used as the final part of the montage[10]. On the next level the betting product structure is found, one folder for each betting product. The next level contains the season folders; summer, autumn, winter and spring. The last level contains the time folders, day and evening. The folder structure can be found in figure 2.4. At the beginning of the thesis work the category folders were at the lowest level of the hierarchy and the betting product folders were at the top level, because it was easiest to add new folders with such hierarchy. This previous folder structure can be found in 2.5. If new folders needs to be added with the current structure, for each new folder the entire hierarchy needs to be created under the folder. The benefits of the current structure are that when a new folder is added, it is only added once, at the top level. With the previous structure the new folder needed to be added at the lowest level of each betting product folder. The main benefit is shown when new vignette types are defined. The most important folder for the user was located at the lowest level, and the other folders were chosen automatically by the system. The hierarchy is then backwards because the user does not need to see the rest of the hierarchy, since it is taken care of by the system.

The audio folder structure is not as complex as the video folder structure. The audio files do not change with the seasons or the time of day. At the top level of the hierarchy are the vignette type folders[11]. Below the top level are the betting product folders, which are the final folders in the audio folder hierarchy.

[10] According to section 1.6 the final part needs to be at least 250 frames long

[11] The vignette types are described in section 1.6.

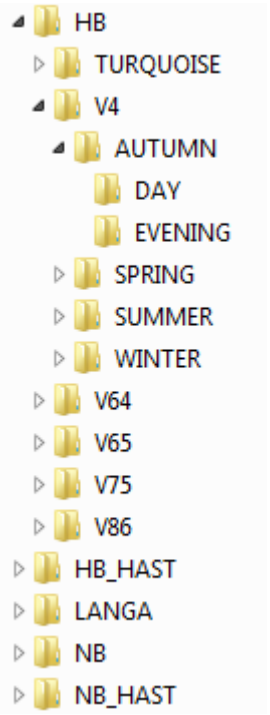

Figure 2.4: The folder hierarchy.

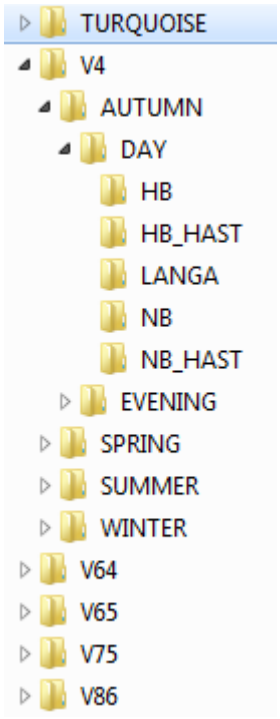

Figure 2.5: The previous folder hierarchy.

XML files

The structure of the vignette system is described in *XML* files. The structure is built up by video definition, audio definition, season definition, time definition, convert process definitions and finally a definition for dummy paths.

The video definition is split into two parts, input video and output video. The input video structure holds information about where the video folder is located and which file extension the video file has[12]. It holds information about which *Avisynth* filters[13] should be applied to the video. It is easy to add new filters or functions in the definition if new filters or functions are needed for the vignettes system. The *XML* structure holds information about, if the video has internal audio, whether that audio be used. If a vignette definition is made where season, betting product and hour of the day is irrelevant, the directory of the folders should be specified without the system adding the folder hierarchy. Therefore the structure has a flag telling whether the folder directory is an exact path or not. If the folders need to

[12]Since the system is going to be as general as possible, it has support for almost every file container.

[13]Read about the different filters at section 2.3.

be ordered by the user, and not randomized by the system, the structure holds information about at which place in the order the current folder should be. If the value is zero, then the system is allowed to randomize the order. The structure allows for adding any number of input video definitions. The second part of the video definition is the output video definition. Only one output video definition is allowed. This definition holds information about where to place the new video file, which name the file will have and what file extension it will have. The file extension will, most of the time, be *MP4*, since that choice was made earlier, but any file extension can be specified. The video output definition also contains information about how many input video files the output video will be built up from. This number does not have to correspond with the number of input video definitions. One input video can be defined, and the output video can be built up by any number of files from that one input video folder. Also the number of input video folders can be larger than the number of video files that define the output video. Lastly the output video contains definitions for which *Avisynth* filters are applied to the output video file. The audio definition is also split into two parts, input audio and output audio. The input audio definition only holds information about the folder path, whether the folder path is an exact path, the file extension and the audio folder's place in order. Currently the place in order definition is automatically set by the vignette system as the same order as the folders are added. No audio filters are available. The output audio definition contains information about how many input audio files the output audio file is built up from. Since the audio will be put together with the video, the audio will end up in the video output file directory. Next comes the season definition. Each month is defined as belonging to a season: summer, autumn, winter or spring. These definitions are rarely changed, but are available if for example a certain year it is snowing in October. October belongs to autumn, but that year it might be changed to winter to get snow footage. The time definition is rarely changed either. It contains definitions about at which hour the day starts and at which hour the evening starts. This is necessary in order for the system to know if the day or evening folder should be chosen. The information about the *ffmpeg* conversion process is defined in the process definition. The *ffmpeg* application's directory, the desired quality and the encoding speed are defined here. Last, and least important, is the definition of the dummy path. This contains information about what file to play while the vignette is being converted. The *Advanced movie clip* node in *Ventuz* has to have a dummy video with which to switch the vignette video, while a new vignette video is being created. Figure 2.6 shows the *Ventuz* output while a new vignette is being created. The progress bar shows the progression of the *ffmpeg* conversion process and the background is the dummy video.

Explained above is the general structure of the vignette system, which is implemented in *Ventuz*. For each new vignette definition created, a new

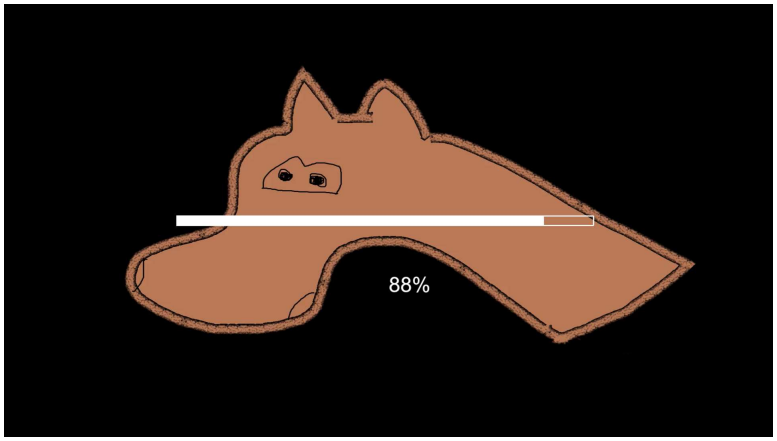

Figure 2.6: *Ventuz* output during the conversion process.

XML file is created which holds the information and definitions of that particular vignette type. When creating a new vignette it is the *XML* file which holds the correct information for the new vignette which is read in *Ventuz*. The user tells *Ventuz* which *XML* file to read through a graphical user interface, see section 2.6.

2.5.2 Ventuz implementation

The *Ventuz* implementation has an event called 'Make vignette'. When this event is triggered[14] the first thing that happens is that a *C#* script gets the path of the *XML* file holding the vignette type definition [15], and tries to open it. If it fails a message is given to the user to check if the path of the *XML* is correct. If the *XML* file was opened successfully, the definitions of the vignette are read and put into the corresponding variables in the *C#* script. Now the system has all the information it needs to build the vignette. The system checks whether there are any video folders in the vignette definition, and if there are the system will start handling the video files. The system checks whether the incoming video folders' directories are exact paths, and if it is not it adds the current betting product, the current season and the current hour of the day. Next the video folders are ordered. For each of the incoming video folders the system checks whether they have a specified place of order. Each place value that is non-zero is stored in a "reserved" positions list. There are three cases that need to be handled when ordering

[14] The event is usually triggered from the GUI described in section 2.6

[15] *Ventuz* gets the path of the *XML* file from another *XML* file, called styrXML, which also contains information about if vignette graphics should be used or not and if so, the definitions for the vignette graphics. StyrXML is created from the GUI, see section 2.6. Current betting product, hour of the day and month are also defined in the styrXML.

the folders. The first case is when a folder has a specified place of order, then that folder is put at its correct position. The second case is when the folder does not have a specified place, but other folders in that definition have specified places. The current folder cannot be placed at the position of another folder, and therefore it is verified that the randomized position of the current folder is not in the list of the "reserved" positions. If it is, a new position is randomly selected. The final case is when there are no "reserved" positions, and the folders can be placed in any order. In this case it is only made sure that the same folder is not used twice. When the folders are ordered they need to be applied to the output video. Two cases are handled here; when the number of input video folders is equal to, or greater than the number of video segments in the final video and when the number of input video folders is less than the number of video segments in the final video. In the first case the folders are ordered in the same way as before. If the number of input folders is greater than output segments the system will take the first folders in the order, and omit the others. In the second case a few, or all, folders have to be used more than once. The order of the folders will be added as many times as needed to fill up the number of output segments. This will prevent the same folder being added twice in a row. The system will try to open each of the folders in the new order and look for files with the specified file extension. One file from each folder is selected at random. If the folder is used more than once, and contains more files than the number of times it is used, the same file is not picked twice. If the same file has to be picked more than once, but the number of files in the folder is greater than one, the same file is not picked twice in row.

A string with the video part of the *Avisynth* script is generated, where the chosen files are opened with their corresponding filters. The other specified input filters are applied to the video files, and the output filters are applied to the output video, which is built from the chosen files.

When the system is done handling the video folders, the system checks whether there are any audio folders in the vignette definition. If there are, the audio folders will be handled. The vignette audio could consist of more than one audio part, and each audio part could have more than one audio file to choose from. If the audio consists of more than one part the different parts have to be put together in a certain order, otherwise the audio output can sound strange. Each audio folder could consist of several audio files to choose from. The system checks whether the audio directory is an exact path, otherwise it adds current betting form to the path. The audio file picker is built in the same way as the video file picker. The audio files will always be placed in the same order as they were added, but the system is implemented the same way as the video files to make it possible to randomize the audio folder order. No filters are added to the audio files at the building of the *Avisynth* audio script part, since no audio filters are implemented.

The system checks whether both the video handling and the audio han-

dling were successful and if they were, the system begins to build the *Avisynth* script using the *Avisynth* script parts from the video and the audio. The system writes the generated *Avisynth* script to an *avs* file and starts an *ffmpeg* process which will convert the *avs* file to an *MP4* file with the *H.264* codec using the *x264* encoder. If there were any errors in the *.avs* file, the *Avisynth* script writes the errors to a log file. The *ffmpeg* process will not know whether there were errors in the script, it only converts the file to *.mp4*. The log file will be read when the process is exited. If there were any errors the system will handle them. Only one *Avisynth* script error is implemented in the system at the moment, because it is the only error encountered during the implementation phase. The error message is "format not supported", and means that the input videos' codec is not installed on the machine. If the system get this error message, all the input video files are converted to *H.264* with *MP4* container, and then the system tries to convert the *avs* file again, from the converted input video files.

When the *ffmpeg* process is started it writes its output to standard error. Standard error can be read inside *C#*. The system reads the standard error and gets information about the duration of the output video file and how many seconds the process has already converted. These two numbers are used to calculate a percentage of the conversion process, to give feedback to the user. If the standard error contains "Permission denied" it usually means that the target video file is in use by another process. In that case the system deletes the file and tries again. If the process takes an unusually long time the process is stopped and started again. Sometimes the process gets into an infinite loop because a child process and its caller are waiting for each other to finish, which results in a deadlock condition. When the process has exited and the log file contains no errors the *C#* scripts triggers an event to *Ventuz*. The event triggers a countdown to when the vignette is going to play. The event also triggers the graphics described in section 3.

2.6 Graphical user interface

To make the vignette system manageable for users with no knowledge of *Ventuz* a graphical user interface was implemented. The interface was developed in *C# .NET*. Two types of user interfaces were implemented to fill different needs. The first was made to enable users to create new vignette definitions. It has functionality to add definitions for input video, add definitions for input audio, change the definitions for the seasons, change the time definitions, change the process definitions, change the dummy path definition and add definitions for output video. The structure of the user interface is implemented to match the structure of the vignette system. The GUI can be seen in figure 2.7. When the user adds a new input video folder the window in figure 2.8 is shown.

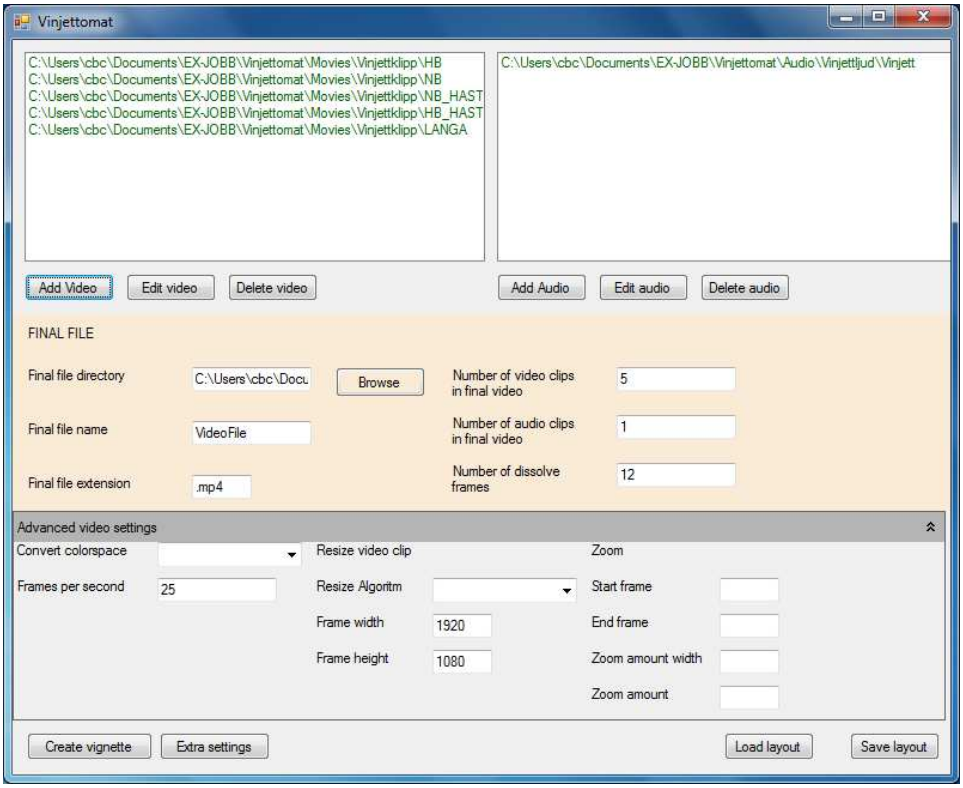

Figure 2.7: GUI for defining a new vignette type

The information about the input video folder[16] is specified here. Under the 'advanced' tab information about the input video filters[17] is specified. Only four filters are available: Convert color space, frame rate, resize and zoom. When the button 'save' is pressed the window will close down and the input video folder will be added to the listview. The input video folder can be edited or deleted. When the user adds a new input audio folder the window in figure 2.9 is shown. The user specifies the information about the input audio folder[18] and then presses the 'save' button. The window is closed and the input audio folder is shown in the audio listview. As with the video, the audio folders can be edited or deleted. When all the video and audio folders have been added[19] the uses specifies information about the output video file[20]. Under the advanced video settings the user can specify the more advanced output video filters. The same filters are implemented as

[16]Explanation about the different input fields can be found in section 2.5.1
[17]Explanation about the input video filters can be found in section 2.3.1
[18]Explanation about the different audio fields can be found in section 2.5.1
[19]As mentioned before, any number of folders can be added.
[20]Explanation about the output video structure can be found in section 2.5.1

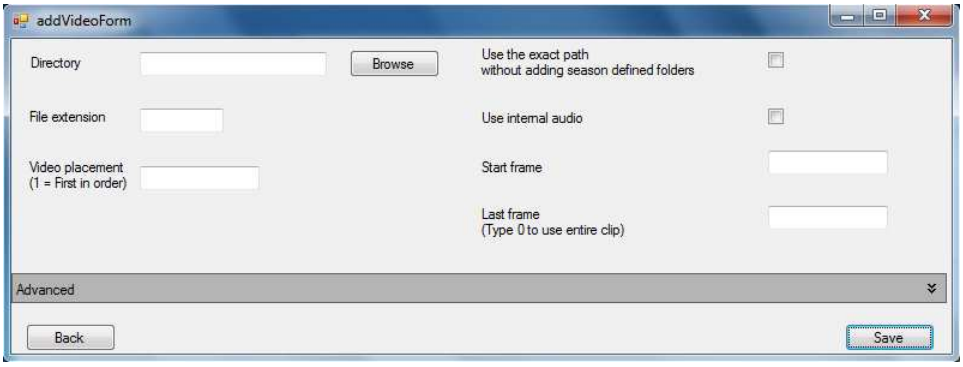

Figure 2.8: GUI for defining the input video folders

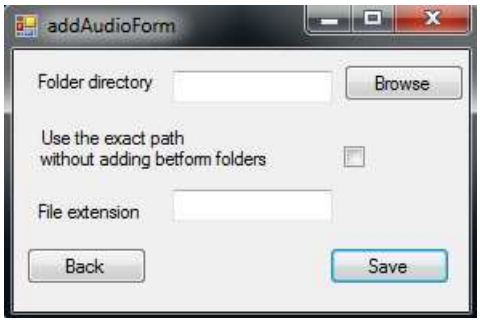

Figure 2.9: GUI for defining the input audio folders

for the input video. When the 'extra setting' button is pressed the window in figure 2.10 is shown. The definitions of the season, time of day, process and dummy path can be changed here. When the new vignette type is finished, the definition is saved as an *XML* file when the 'save' button is pressed. The user specifies a folder, either a new folder or an already existing folder, and the *XML* file with the new vignette definition is saved in the folder under the same name as the folder. The new vignette type is named the same as the *XML* file containing the definition. If a new vignette type is to be defined which is similar to another vignette type, the user can press the 'load layout' button and load a previous defined vignette type. The 'create vignette' button is pressed if the user wants to see an example of a vignette of the new vignette type. Another type of graphical user interface, which is described in the following paragraph, is shown.

A smaller graphical user interface was also created. It is used when creating new vignettes of a certain type, not when creating new vignette definitions. There are different types of this interface, the first one is for

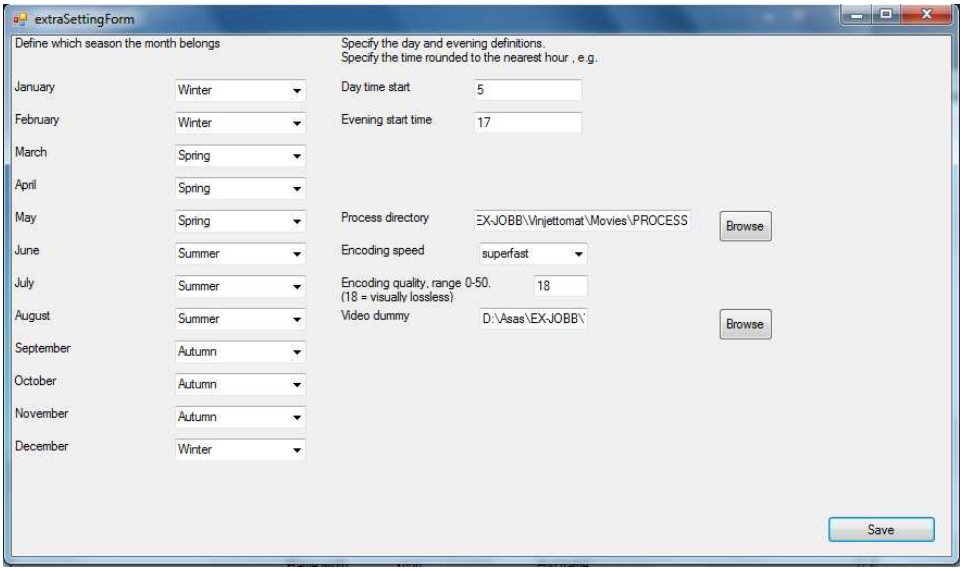

Figure 2.10: GUI for changing the season, time, process and dummy definitions

creating a vignette with graphics. The user specifies which text layer[21] to use, the text, the betting product, the time and date the vignette is made for and which type of vignette is to be made. Figure 2.11 shows the small graphical user interface.

When the 'Make vignette' button is pressed the interface saves the specified information in an *XML* file, called styrXML, which *Ventuz* reads. The *XML* file contains a definition of whether the vignette includes graphics or not. Using this type of interface that flag is set to true. *Ventuz* will get the information from the *XML* file about which vignette definition to read, the current betting product, date and time. The graphics part of *Ventuz* gets information about which text layer to use, the text and the betting product. When the 'Make vignette' button is pressed *Ventuz* also get the information to trigger the create vignette event. If the button 'Play vignette' is pressed *Ventuz* triggers the play vignette event, which plays the created vignette again without making a new vignette. The second type of this interface is for creating vignettes without graphics. It is similar to the previous type, but does not contain input for text layers or text. The definition if the vignette contains graphics is set to false. Figure 2.12 shows the small graphical user interface without graphics.

[21] see section 3.5

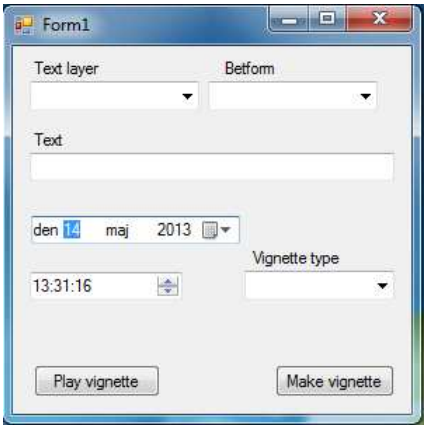

Figure 2.11: GUI for creating new vignettes of a certain vignette type.

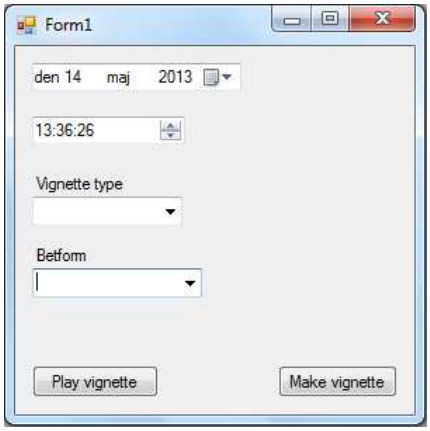

Figure 2.12: GUI for creating new vignettes of a certain vignette type without adding graphics.

Chapter 3

Vignette graphics

When the vignette video is created it is imported into the previous mentioned Ädvanced video clipñode. The video is rendered together with the graphics described in this chapter with real-time rendering when the vignette is played.

3.1 ATG's logotype

A year ago ATG changed their graphical profile. Their new logotype consists of 22 parallelograms, put together into a plate, see figure 3.1. Their vision was that these parallelograms could be animated, and be symbolic of racing horses. The work on creating these animated parallelograms in *Ventuz* was started in the autumn of 2011. The parallelograms were finished that same year, but they sometimes had performance issues. A part of this thesis work was to find ways to optimize the parallelograms to give better performance. Another part of this thesis work was to implement the parallelograms in the vignette system.

Figure 3.1: ATG's logotype

3.1.1 Logotype implementation

The 22 parallelograms are built up from two cubes. Using an *HLSL shader* node these two cubes are each divided into 11 parallelograms. Inside the shader the vertices are moved to be on top of each other to create the smaller parallelograms. Figure 3.2 shows how one rectangle is split into two

rectangles by moving the vertices. The same method is used to split one cube into 11 cubes.

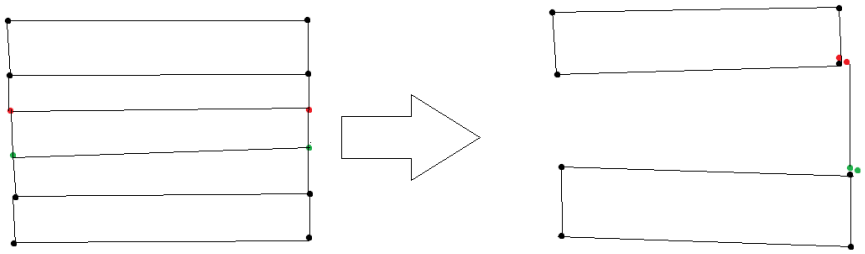

Figure 3.2: Splitting rectangles into smaller rectangles by moving vertices.

Each parallelogram contains eight vertices[1]. Inside the shader the parallelograms are scaled, moved and colored. All the information about the parallelograms can be found in an *Excel* document, which is read inside *Ventuz*, see appendix B.1. The first column in the *Excel* sheet corresponds to the x position of the parallelogram. The second column corresponds to the y position of the parallelogram. The width and the height of the parallelograms are described in column 3 and 4. Column 5 holds information about how much the top four vertices are offset from the bottom ones to create the parallelogram geometry instead of a rectangle. Columns 6, 7 and 8 are the first RGB values of each parallelogram, and columns 9, 10 and 11 are the second RGB values of each parallelogram. A gradient is drawn on each parallelogram between the two color values. The transparency of the parallelograms will be a value in between the values in column 12 and 13[2]. The animation of the parallelograms is state-based and each sheet in the *Excel* document corresponds to a state. *Ventuz* reads the *Excel* file and places the information about the parallelograms in arrays. All the values from column one in the first sheet go into one array and all the values from column one in sheet two, or state two, go into another array. This is done for all columns. The parallelogram pattern has two events, start and stop. When the start event is triggered, *Ventuz* uses linear interpolation between the values of

[1]They consist of 12 vertices, but the extra vertices are placed on top of the other vertices

[2]The transparency ranges from 0 - 100, where zero is fully transparent and 100 is fully opaque.

the two states. In most cases it is only the x-value of the parallelograms which changes. For the x-values' linear interpolation, a node called *Mover* is used. The *Mover* is used because of its built-in functionality. The *Mover* has a parameter 'mode' which is the main reason it was chosen. 'Mode' has four different selections. The three of them used in this implementation are Absolute, One shot and Infinite. Absolute interpolates between the two values without needing a trigger. As soon as the node is created it starts to interpolate and goes on forever. One shot has to be triggered to start and when triggered it interpolates between the two values once. Infinite also has to be triggered to start interpolating, but does not stop interpolating until the stop event is triggered. The duration of the interpolation is based on the size of the parallelogram. Bigger parallelograms move more slowly than smaller parallelograms. The parallelograms are categorized into big, medium and small parallelograms. Each category has its own duration interval with a maximum duration and a minimum duration. Each parallelogram gets its own duration since a duration value is randomized within the interval. Each time the parallelogram has finished one interpolation and is back at state one a new duration value is calculated. When using the state based internal movement the parallelograms "jump" back to their starting position when the interpolation is done. To make the "jump" look smoother the parallelograms are faded from zero to their transparency values when they start from the first state again.

As mentioned earlier the shader takes care of scaling, moving and coloring the parallelograms. The output values of the interpolation are passed to and handled by the shader. The shader also has parameters for moving, scaling and rotating the parallelogram pattern as a unit.

The optimization of the parallelograms consisted of making better node choices and optimizing the shader code. Before, the *HLSL* input values were stored in separate variables. By putting the values of the same category, such as x positions, in arrays instead, the shader code became much more efficient. Instead of having several *C#* scripts with different functionality they were all put into one script. For every node added, *Ventuz* has to evaluate that node, which means that fewer nodes leads to better performance.

3.1.2 Logotype implementation in the vignette system

In the implementation of the vignette system the parallelogram pattern is used to swoop over the screen, as shown in to figure 3.3, when there is a dissolve between two video clips.

The parallelograms need to move internally within the logotype to keep the impression of racing horses and, at the same time move as a unit across the screen. For the internal movement the state-based animation described in the previous section is used. For the movement across the screen the shader parameter for moving the parallelogram pattern in the x direction is

Figure 3.3: The parallelogram pattern swooping across the screen

changed. When the parallelograms swoop the screen they need to be able to curve smoothly. To enable smooth curving for the parallelogram pattern, the pattern is put in a *Render target* node. The *Render target* saves the rendered output from the elements which are put after the *Render target* node, see figure 3.4, and provides it as a texture. The saved output from the *Render target* is used as a texture on a plane which is larger than the screen. An *HLSL shader* node is used to curve the big plane using sine and cosine functions. When the parallelogram pattern moves in the x direction on the curved plane, it will look as if the parallelogram pattern is curved. Listing 3.1 shows the equation used to move the plane's vertices and get a curved surface.

```
float p = (Position.x + Position.y + Position.z) + time  * 3.141592653f;
float p2 = (Position.x + Position.y + Position.z) + timeMove  * 3.141592653f;
// Add some oscillation
Position.x += sin(p * frequency.x) * cos(p * frequency.x) * amplitude.x / 10.0f;
Position.y += sin(p * frequency.y) * cos(p * frequency.y) * amplitude.y / 10.0f;
Position.z += cos(p2 * frequency.z) * amplitude.z / 10.0f;
```

Listing 3.1: HLSL code for curving the plane

The amplitude, frequency and time parameters are changed for each parallelogram swoop as well as the swooping angle, which contributes to making each vignette unique. The parameters are randomized in a *C#* script. Figure 3.5 shows the *C#* script and the *HLSL* node in the content window, and how the *HLSL* node takes the generated parameters from the *C#* script as input parameters. Figure 3.5 also shows how the *HLSL* node takes the *Ren-*

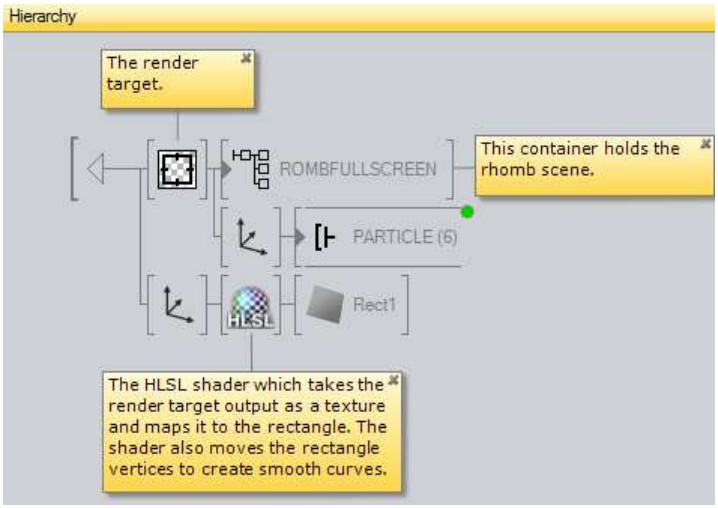

Figure 3.4: The *Render target* node in *Ventuz*. Where it says *rhomb* or *romb*, parallelogram is meant.

der target as input. Because they are randomized, the swoops can sometimes look strange. Therefore a process of finding good looking swoops was set up where the *C#* script randomizes the parameters and if the parallelogram swoop was acceptable, the parameters of the swoop were saved in a text file. The swoops are saved in different text files depending in their appearance. Similar looking swoops are saved in the same text file. The vignette system reads the text files and chooses swoops at random from the text files. One swoop from each text file is used to get a variety of swoops. This ensures the quality of the vignettes.

There is one swoop for each dissolve between video clips, except for the last dissolve. The *Mover* node switches mode from absolute to infinite and the amplitude and frequency parameters are set to zero to make the plane flat again. When the final video clip starts playing the parallelograms' start event is triggered and the parallelograms enter the screen, covering about a third of the screen. Only the internal movement of the parallelograms is used. The parallelograms work as a background for the text layer[3], according to figure 3.6.

3.2 HLSL blend modes

When the parallelograms swoop across the screen they are moving quickly and are blended with the background since they are not fully opaque. To make sure that focus lies on the parallelograms, the *Photoshop* blending

[3]see section 3.5

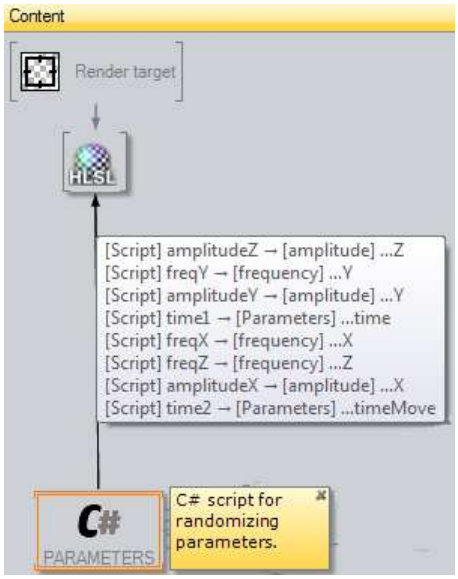

Figure 3.5: The connection in *Ventuz* between the *HLSL shader* node, the *C#* script and the *Render target*

modes are implemented in *Ventuz*. The blending modes are implemented in an *HLSL shader* node and the code is brought from [23]. No changes were made to the code. Explanation about the blending modes can be found in [24]. For the parallelogram pattern the blending mode color dodge is used. The shader takes the parallelogram pattern as a texture as input together with another texture, figure 3.7. These two textures are blended together according to the specified blending mode.

3.3 Particles

A part of this thesis work was to implement a particle system in *Ventuz*. The particles were to be emitted from the position of the parallelogram[4] leaving a trail as the parallelogram, pattern swoops over the screen. The effect of the particles is to add extra focus to the parallelogram patterns, and give a more exciting look to the vignettes. There is a commercial particle system sold by a company named Glare Technologies. From their website [25] a demo of the particle system, which will be referred to as *Glare's particle system*, can be downloaded for trial. Before starting the work of creating a particle system, *Glare's particle system* was tested to see if it could be used instead. *Glare's particle system* has several parameters, such as emission point, number of particles, particle color, etc. The essential disadvantage was that the

[4]The parallelograms from section 3.1

Figure 3.6: The parallelogram pattern together with a text layer

particle system's internal coordinate system could not be accessed. Because of that disadvantage the particles could not follow the same path as the parallelogram patterns. Another disadvantage is the lack of complete control over the particles, which could be gained by creating a new particle system. *Glare's particle system* emits their particles from either a cube or an ellipse, and by creating a new particle system the particles can, for example, be emitted from a mesh. Another advantage of building a particle system was that a simple particle system had already been built which could be used as a starting point. The decision was to build a new particle system which could be completely controlled. Several tutorials were studied to see if they were applicable on a particle system in *Ventuz*. Most of them turned out to not be very helpful. The biggest problem was, and still is, that *Ventuz* does not allow users to create geometry dynamically. If it did, it would probably cause problems for the real-time rendering. This means that for each particle in the particle system one particle geometry has to be dragged into the interface and therefore a limitation of a thousand particles is set. The particle system's parameters were decided to be; number of particles, minimum and maximum spreading angle, minimum and maximum speed, minimum and maximum scaling factor, starting color, target color, emission point and emission area. A *C#* script was used to calculate the emission points of the particles, and also to produce random values. *HLSL* shader model above 1.0 cannot produce random values, instead they are created in the *C#* script and saved in each particle's *Material* node. The *HLSL shader* node makes *Ventuz* ignore the appearance nodes which are added after the shader node.

Figure 3.7: Blending glow texture, used together with the parallelogram pattern for blending. The glow is white in the implementation, but is made grey to make it visible in the thesis report.

The *Material* nodes can, however, be read inside the shader and the shader therefore gets access to the random values. The emission points are saved in *World* nodes, instead of *Material* nodes. *Material* nodes cannot store negative values, and the system needs to be able to emit particles at negative x- and y positions. Inside the shader the particles' velocities, positions, scaling and colors are calculated and updated. The angle at which the particles are emitted is calculated using the specified minimum and maximum angle values, the random values from the *C#* script are used to calculate random angles within the given angle range. Each particle has its own angle value. The speed of the particles is calculated using the specified minimum and maximum speed. The velocity of the particles is calculated as the speed multiplied by the cosine of the angle for the velocity in the x direction, and the sine of the angle for the velocity in the y direction. The velocities are added to the positions of the particles to update the positions. The scaling of the particles is also a randomized value between the maximum and minimum scale values.

The particle system in the vignette system only has 200 particles. This gives better performance and the number is enough to add focus to the parallelograms. Figure 3.8 shows the *Ventuz* implementation of the particles.

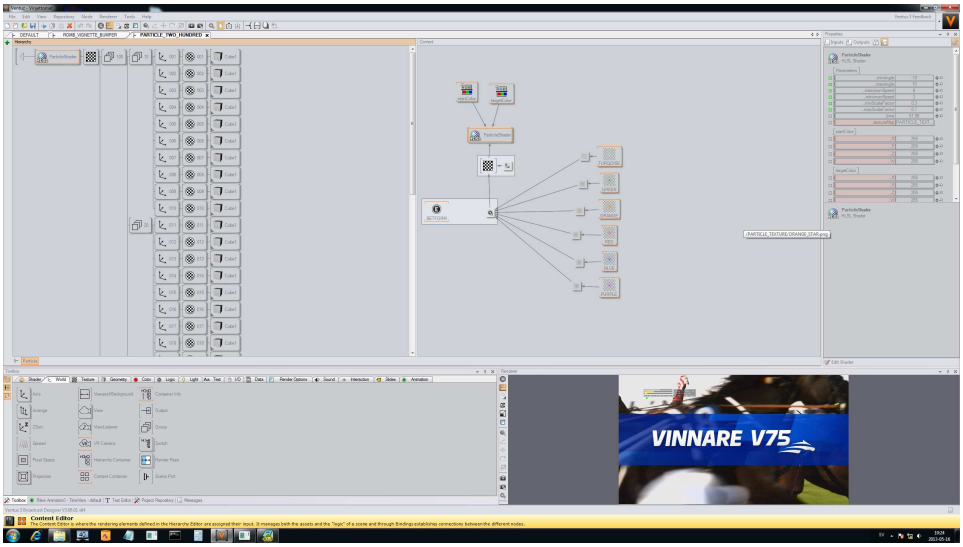

Figure 3.8: The particle implementation in *Ventuz*

3.4 Motion Blur

Motion blur is a widely used effect for adding a feeling of speed in graphics. Motion blur was implemented in *Ventuz* for the vignettes system's graphics to be used on the text layers, see section 3.5. The effect was implemented in an *HLSL shade*r node. The implementation in *Ventuz* is based on the current object's position and previous position. A *C#* script takes the object's x and y positions as input and stores these values. Each time a frame is rendered the new input positions are compared to the stored, previous positions. The difference between the two corresponding values is calculated and the total movement of the object is calculated using the Pythagorean theorem. A blurring factor is multiplied with the total movement to compute the current amount of motion blur. The angle of the movement is also calculated and passed to the *HLSL shader* node together with the blur amount value. Inside the shader, which reads the object as a texture and converts it into pixel color values, the object is rendered multiple times [5]. At each rendering the object is shifted according to the angle and the blur amount. This gives the effect of motion blur.

3.5 Textlayer

At the end of the vignettes a text is overlaid on top of all the other graphics. This text informs the viewers of what will be shown next. Some texts say

[5]In this implementation the objects is rendered 25 times

which TV show will follow after the vignette, others lets the viewer know which race will be shown. Some vignettes explain which topic will be covered next. These text layers may also contain a logotype. Each betting product has its own logotype. *Elitloppet* and *Olympiatravet* are also examples of races with their own logotype. For this thesis work six different types of text layers were implemented.

The text font used is a specially designed font for ATG, which is available in 3D, for *Ventuz* usage. Each of these layers, except for the last one, has an animation with 4 states. The first two states describe how the text enters the screen from the left side into the end position. The second two states describe how the text enters the screen from the right side to the end position. Motion blur, see section 3.4, is used on the text animation.

3.5.1 First layer

The first text layer consists of simple text without logotypes and with every word being of the same size. The challenge with simple text is to ensure that the text lies within the borders of the graphics safe area, see figure 3.9, the text has to be shrunk to keep within the borders.

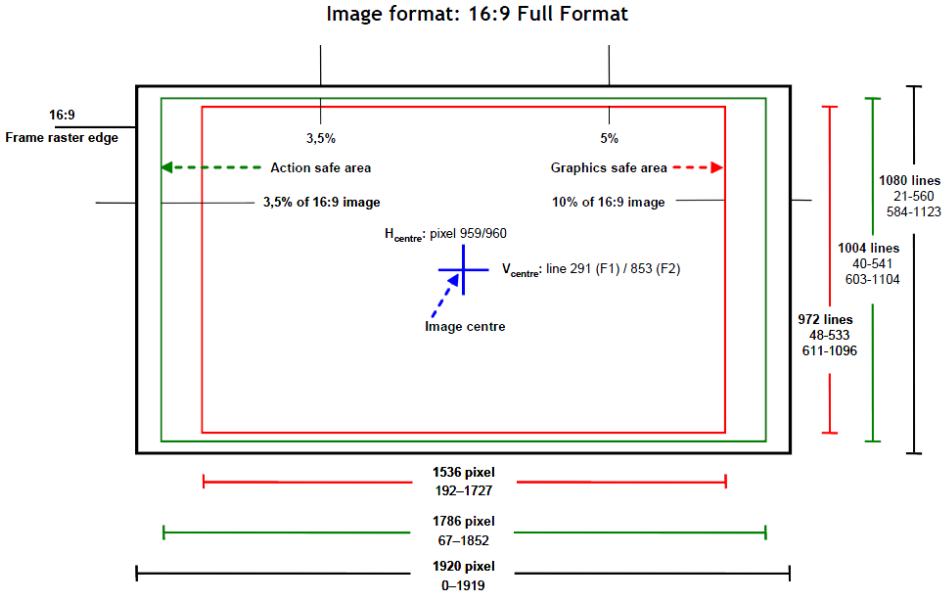

Figure 3.9: Television safe areas for 1920x1080 resolution, from [12]. For more information about safe areas, see [12].

However if the text gets shrunk too much it will become unreadable. If the text passes the threshold of unreadability it is split into two rows. To

avoid having widows[6] in the text an algorithm was implemented to make the text evenly distributed between the two rows. The *Ventuz Block text* node, which is used for text in this implementation, has built in functionality for splitting text into two rows if the length passes a certain width. This width is used to evenly distribute the text on the two rows. The width is calculated according to listing 3.2. In the expression A is the value for the graphics safe area in figure 3.9. B is the width of the text and finally C is the width of the second last word of the text.

```
if(B > A)
{
  if((B/2) + C) > A)
  {
    return A;
  }
  else
  {
    return ((B/2) + C);
  }
}
else
{
  return A;
}
```

Listing 3.2: Expression for calculating the width used to evenly distribute the text on two rows. The returned value corresponds to the width.

3.5.2 Second layer

The second text layer is a combination of text and a logotype. The text and logotype together have to be centered on the screen. An algorithm is implemented to calculate where the text and the logotype will be placed depending on the length of the text and the size of the logotype. The width of the logotype and the width of the text are added together with the width of the space between them. The sum of the widths is subtracted from the width of the screen in the *Ventuz* scene, and the result is divided by two. This will give the width of the space on each side of the text and logo together. The text's left side should therefore start at that value. The texts and logotypes are center aligned and the origin of the *Ventuz* coordinate system is at the center of the screen. The text's x position is calculated according to 3.1. A is the value of the space before the text begins. B is

[6] A single word which stands alone on the upper or lower row

the negative part of half the width of the screen. C is the width of the text. Half the text width is added because the text is center aligned.

$$(B + A) + (C/2) \tag{3.1}$$

The logotype's x position is calculated according to 3.2. A is the width of the text. B is the space between the text and the logotype. C is the result of 3.1 and D is the width of the logotype.

$$C + (A/2) + B + (D/2) \tag{3.2}$$

Both the text and the logo need to have their own motion blur shader, because their animations will be triggered at different times. The logotype's animation is triggered first, and the text's animation is triggered 10 frames later.

3.5.3 Third layer

The third text layer is similar to the second one, but the logotype is placed at the text's left upper corner. The computation of this layer is the same as for the second layer, but the logotype and the text have switched places.

3.5.4 Fourth layer

The fourth layer is a text with two rows, where the different rows can have different font sizes. The two rows have their own animations and therefore their own motion blur implementations. The first row's animation is triggered first and 10 frames later the second row's animation is triggered. The texts are not allowed to grow beyond the borders of graphics safe area, see figure 3.9. A message is given to the user if the texts are too long. Each of the two rows is implemented in the same way as the first layer, but the texts are not able to split into two rows. If the texts were able to split into two rows, the vignette could end up with four rows in the worst case.

3.5.5 Fifth layer

The fifth layer consists of an imported image made outside *Ventuz* which will be used instead of a text. This layer is chosen when the above described layers do not meet the requirement of the text, such as if both a logotype and a two row text is needed, or the placement of the logo or the text needs to be changed. The image needs to be made in the same resolution as the vignette video. Motion blur is applied to the image during the animation.

3.5.6 Sixth layer

The sixth and last layer consists of an animation made outside *Ventuz*. This is the simplest layer. It only has the parameters play and reset. This layer can be used when none of the other layers meet the requirements of the text, for example if the logotype is animated. The animation needs to be made in the same resolution as the vignette video.

Chapter 4

Conclusions and future work

4.1 Conclusion

The requirements for the vignette system were that a new vignette could be made one hour before broadcast, and in a maximum of 15 minutes. The process of converting the *Avisynth* file, with the *ffmpeg* parameters[1] chosen to be visually lossless for quality and 'superspeed' for encoding speed, is approximately real time. Converting in real time means converting the file takes about the same time as the duration of the video file. A vignette is about 15 - 20 seconds long. The time for setting up a new vignette definition is not included in the 15 minutes time, because setting up new definitions is not needed every time a new vignette is created. However, an experienced user could easily define a new vignette and create the vignette in 15 minutes. With the 15 minute time frame it is possible to produce several vignettes of the desired type, and choose the one which is most appealing.

The system produce vignettes of good quality, and the graphics do not freeze. However, the *ffmpeg* process sometimes crashes in the 32-bit version. The 32-bit version is used when the file format of the input video files is *QuickTime*. Since *QuickTime* is only available in 32-bit, *ffmpeg* 64-bit is unable to convert *QuickTime*. There is a restriction of how many input video files *ffmpeg* 32-bit can handle without crashing. For the vignettes described in this thesis *ffmpeg* works most of the time. If the process crashes or freezes, the users are trained to restart the process. Since the video montage is done before broadcast, and the process requires only a few seconds, this solution is acceptable.

During the implementation of the system no file format was found that the system was unable to handle. This is an important part of the system's generality. This opens up the possibility to later choose a different, more efficient codec or container format.

The vignette system has already been used to produce vignettes for

[1] See section 2.4.

Elitloppet. Vignettes for the betting product, *V75*, with summer footage have also been produced. It was obvious when producing the vignettes that the system saves a lot of time, and that the system produces vignettes of good quality.

4.2 Future work

4.2.1 Codec and Container

As new codecs and containers are developed continuously, future work could involve finding better codecs and containers to get even better performance. *Matroska, .mkv*, a container format which was discovered during the development, is one container format which sounds interesting and should be investigated further. Finding another work flow for gathering video files which do not need the *QuickTime* container format would also be preferred, to avoid the need for using the *ffmpeg* 32-bit version, due to its instability.

4.2.2 Avisynth and GUI

The *Avisynth* part of the thesis work could be developed further by implementing all the internal filters and the already implemented filters could be developed further by adding all the available parameters. Not all of the implemented filters are available in the graphical user interface so the GUI could be improved by adding the missing *Avisynth* filters.

4.2.3 Particles

Future work on the particle system would be to make it possible for the particles to be emitted from geometries, and not only points and quadrangles. A way to do this would be to make the particles be emitted evenly spread from the boundary vertices of the current geometry.

Appendices

Appendix A

Avisynth appendices

A.1 Avisynth example script

```
# This is a comment in Avisynth scripting
# The following line opens the file ”video1.mp4”
# and converts it to color space YV12. According to the
# trim function the video will play between frames 0 and 60
video1 = DirectShowSource(”video1.mp4”).ConvertToYV12.Trim(0, 60)
video2 = DirectShowSource(”video2.mp4”).ConvertToYV12.Trim(0, 60)
# The two video clips are put together with a 12 frame dissolve
VideoOut = Dissolve(video1, video2, 12)
# The three following lines are for deinterlacing the video.
VideoOut = SeparateFields(VideoOut)
VideoOut = Bob(VideoOut)
VideoOut = SelectEven(VideoOut)
# The audio is opened.
audio = DiretShowSource(”audio.wav”)
# The audio file and the video are put together
AudioDub(VideoOut, audio)
```

A.2 Zoom filter

```
#This function enlarges the video according to the
#previously calculated new frame sizes, and then crops
# the video back to its original frame size.
Function myfunc(clip c1, int x, int y, int w, int h)
{
  test = c1.BilinearResize(x,y)
  test = test.Crop((x−w)/2,(y−h)/2, w , h)
  return test
}
#This is my implementation of the zoom filter in Avisynth.
#It takes the video clip, the video frame on which the zoom should begin,
#the video frame in which the zoom should end, the vide frame width,
#the video frame height,
#the number of times the video should be zoomed in the x direction and the number
#of times the video should be zoomed in the y direction.
```

```
Function zoom(clip video, int startFrame, int endFrame, int frameSizeX,
int frameSizeY, float zoomAmountX, int zoomAmountY)

  #Calculate the new frame sizes for the zoom.
  zoomX = frameSizeX * zoomAmountX
  zoomY = frameSizeY * zoomAmountY
  #Animate is an internal filter in Avisynth. It linearly interpolates the arguments
  #between startFrame and endFrame.
  video = Animate(startFrame, endFrame , "myfunc", video.ConvertToRGB(), frameSizeX, frameSizeY,
  frameSizeX, frameSizeY, video.ConvertToRGB(), zoomX,  zoomY, frameSizeX, frameSizeY)
  return video
}
```

Appendix B

ATG's logotype appendices

B.1 Excel document

Microsoft Excel (Product Activation Failed) - Blå

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	-3,3614	0	0,37106	1,085189	0,003904	0	32	147	0	86	224	80	95
2	-2,514	-0,4839	0,272427	2,226129	0,008009	0	32	147	0	86	224	80	95
3	-0,7348	-0,2873	0,077795	1,160387	0,004172	0	32	147	0	86	224	80	95
4	-2,2827	-0,3768	0,249637	1,388575	0,004997	0	32	147	0	86	224	80	95
5	-1,3995	-0,3327	0,146309	2,722697	0,009805	0	32	147	0	86	224	80	95
6	-2,8189	-0,3327	0,307387	1,960342	0,007061	0	32	147	0	86	224	80	95
7	-1,6975	0	0,173839	4,31353	0,015517	0	32	147	0	86	224	80	95
8	-1,9503	-0,2873	0,205642	3,311318	0,011914	0	32	147	0	86	224	80	95
9	-2,463	-0,1664	0,266983	2,156117	0,007764	0	32	147	0	86	224	80	95
10	-2,0562	-0,5428	0,224104	1,463774	0,005265	0	32	147	0	86	224	80	95
11	-2,5558	0	0,281195	1,085189	0,003904	0	32	147	0	86	224	80	95
12	-2,4353	0	0,25613	4,31353	0,015528	0	32	147	0	86	224	80	95
13	-3,3673	-0,2334	0,361198	4,010143	0,014423	0	32	147	0	86	224	80	95
14	-1,7632	-0,1664	0,186868	2,727883	0,009816	0	32	147	0	86	224	80	95
15	-3,09	-0,4839	0,339368	1,478035	0,005321	0	32	147	0	86	224	80	95
16	-2,8467	-0,5428	0,312284	1,463774	0,005265	0	32	147	0	86	224	80	95
17	-2,7449	0	0,298418	2,157413	0,007775	0	32	147	0	86	224	80	95
18	-2,2258	-0,0837	0,238784	2,639719	0,009504	0	32	147	0	86	224	80	95
19	-1,221	-0,0837	0,126698	2,639719	0,009504	0	32	147	0	86	224	80	95
20	-2,8886	-0,4839	0,314214	2,226129	0,008009	0	32	147	0	86	224	80	95
21	-2,1601	-0,3768	0,233217	2,15093	0,007742	0	32	147	0	86	224	80	95
22	-2,153	0	0,217578	6,273871	0,022589	0	32	147	0	86	224	80	95
23													
24													
25													
26													
27													
28													
29													
30													
31													
32		55,28				896,46	65,56						
33		54,51											
34													
35													

State 1 | State2 | OriginalMönster | State3 | State4 | State5 | State6 | State7

Figure B.1: *Excel* document with information about the 22 parallelograms

Bibliography

[1] AB Trav och Galopp, ATG, www.atg.se, 2013. [Online; accessed 14-May-2013]

[2] AviSynth developers and contributors, Avisynth, http://avisynth.org. 2012. [Online; accessed 07-May-2013].

[3] AviSynth developers and contributors, Avisynth, http://avisynth.org/mediawiki/Internal_filters, 2013. [Online; accessed 07-May-2013].

[4] AviSynth developers and contributors, Avisynth, http://avisynth.org/mediawiki/ConvertToRGB24, 2013. [Online; accessed 07-May-2013].

[5] AviSynth developers and contributors, Avisynth, http://avisynth.org/mediawiki/BilinearResize, 2011. [Online; accessed 07-May-2013].

[6] AviSynth developers and contributors, Avisynth, http://avisynth.org/mediawiki/Weave, 2013. [Online; accessed 07-May-2013].

[7] AviSynth developers and contributors, Avisynth, http://avisynth.org/mediawiki/Bob, 2007. [Online; accessed 07-May-2013].

[8] AviSynth developers and contributors, Avisynth, http://avisynth.org/mediawiki/Overlay, 2013. [Online; accessed 07-May-2013].

[9] FFMPeg. FFMPeg. http://www.ffmpeg.org/, 2013. [Online; accessed 07-May-2013].

[10] FFMPeg, FFmpeg and x264 Encoding Guide, http://ffmpeg.org/trac/ffmpeg/wiki/x264EncodingGuide, 2013. [Online; accessed 09-May-2013].

[11] Gary J. Sullivan et. al. "The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions" http://www.fastvdo.com/spie04/spie04-h264OverviewPaper.pdf, SPIE Conference on Applications of Digital Image Processing XXVII, August, 2004.

[12] MPEG. The Moving Picture Experts Group website, http://mpeg.chiariglione.org/standards/mpeg-4/advanced-video-coding, 2013. [Online; accessed 07-May-2013].

[13] Iain E. Richardson John Wiley & Sons *The H.264 Advanced Video Compression Standard* 2010, Second edition.

[14] Avid Systems. Avid DNxHD Technology, http://www.avid.com/static/resources/US/documents/DNxHD.pdf, 2013. [Online; accessed 07-May-2013].

[15] MPEG, The Moving Picture Experts Group website, http://mpeg.chiariglione.org/standards/mpeg-4, 2013. [Online; accessed 07-May-2013].

[16] Peter Lambert et. al. "Rate-Distortion Performance of H.264/AVC Compared to State-of-the-Art Video Codecs", IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, VOL. 16, NO. 1, JANUARY 2006

[17] Xvid Systems. Xvid video codec. http://www.xvid.org/General-Info.19.0.html, 2013. [Online; accessed 07-May-2013].

[18] DivX Inc. DivX video codec. http://labs.divx.com/node/4904, 2012. [Online; accessed 08-May-2013].

[19] Dr. D.Vatolin et. al. Video Group of MSU Graphics and Media Lab, http://www.compression.ru/video/codec_comparison/h264_2010/

[20] Sarnoff Corporation. Princeton, NJ. "Measuring Image Quality: Sarnoff's JNDmetrix Technology" Jul. 2002.

[21] T. Wiegand et. al, "Rate-constrained coder control and comparison of video coding standards" IEEE Trans. Circuits Syst. Video Technol., vol. 13, no. 7, pp. 688-703, July 2003.

[22] Ventuz Technology Group, Ventuz, http://ventuz.com/products/designer.aspx. 2013. [Online; accessed 06-May-2013]

[23] Romain Dura, Photoshop math with HLSL shaders, http://mouaif.wordpress.com/2009/01/08/photoshopmathwithhlsl-shaders/ 2009. [Online; accessed 13-May-2013]

[24] Adobe Systems Incorporated, Photoshop Blending Modes, http://help.adobe.com/en_US/photoshop/cs/using/WSfd1234e1c4b69f30ea53e41001031ab64-77eba.html 2013. [Online; accessed 13-May-2013]

[25] Glare Technologies, Glare Technologies, http://www.glare-technologies.com/shop/ 2013. [Online; accessed 13-May-2013]